TF-TRT BEST PRACTICE, EAST AS AN EXAMPLE

Xiaowei Wang (王晓伟), Dec 18th, 2019

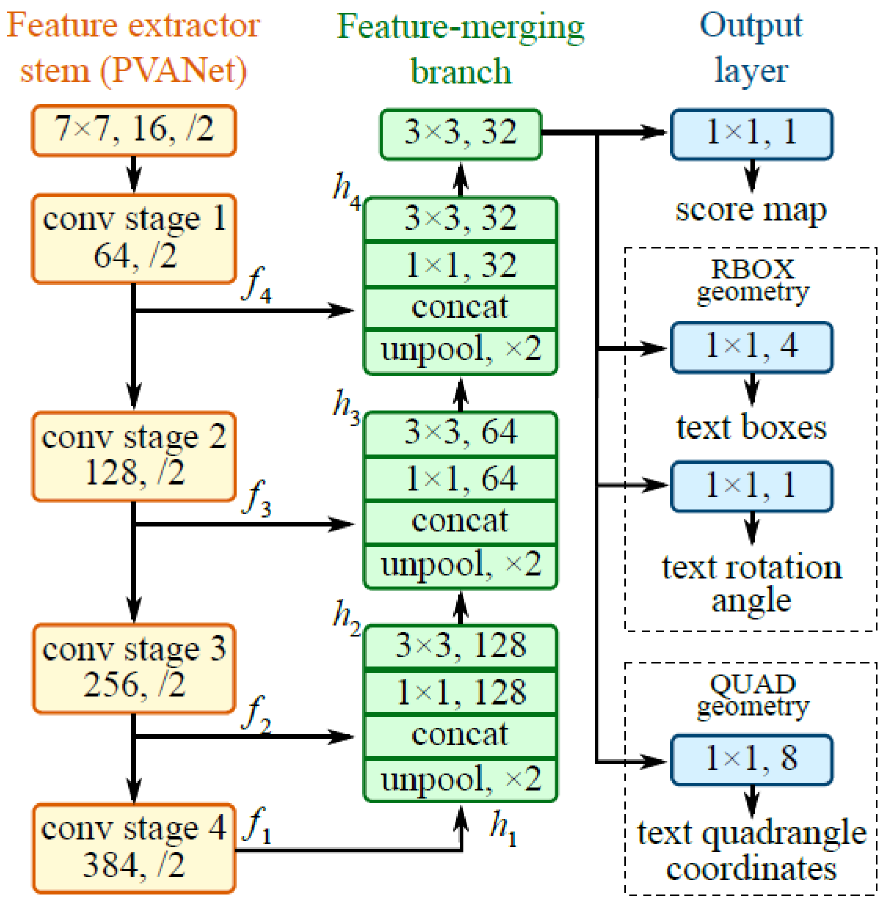
OUTLINE

- Background
- TFTRT
- TRT API
- TRT UFF Parser
- Conclusion

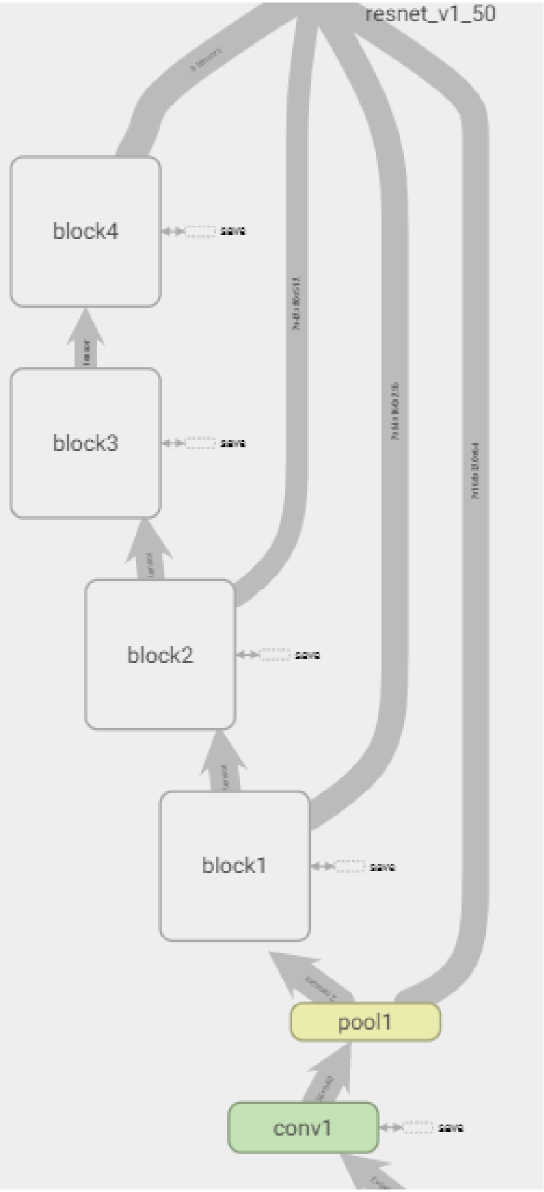
BACKGROUND

EAST for Ali

A fully-convolutional network (FCN) adapted for **text detection** that outputs dense per-pixel predictions of words or text lines.



https://arxiv.org/abs/1704.03155

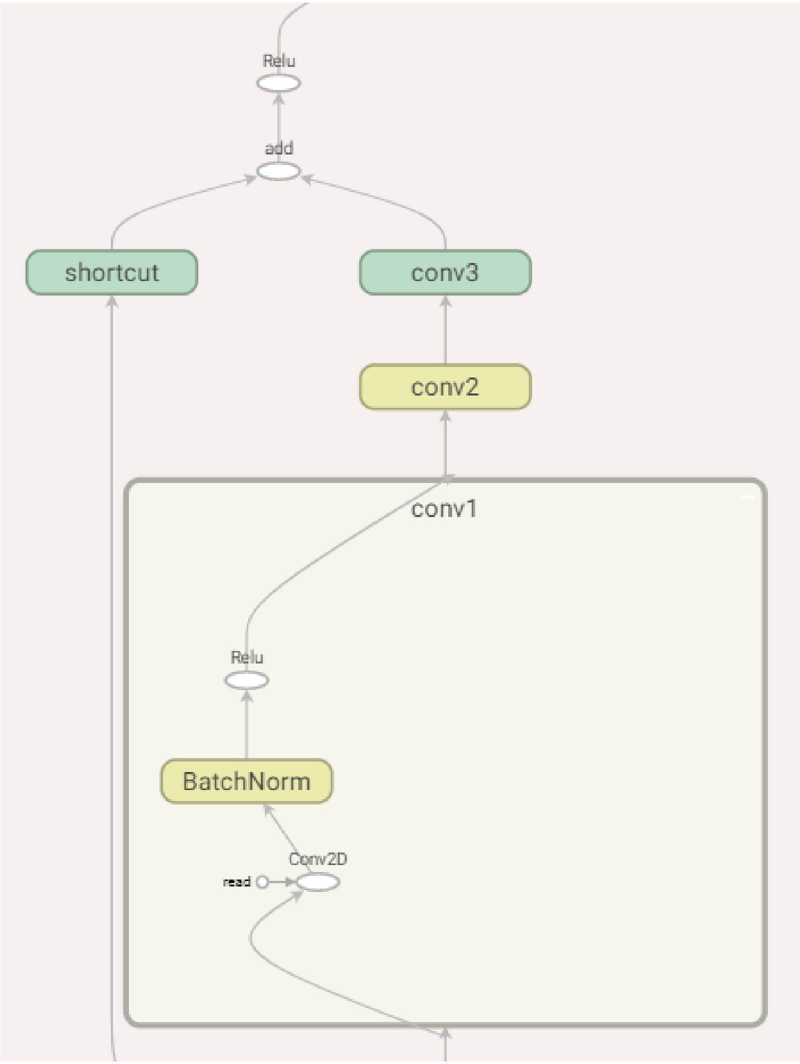


https://github.com/argman/EAST

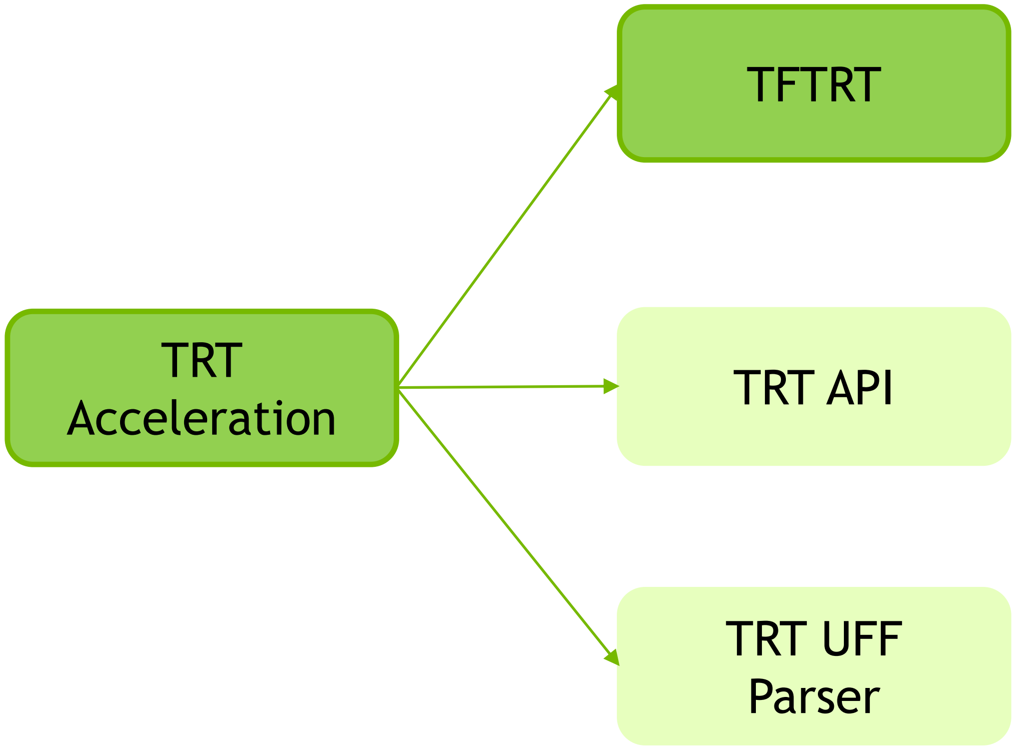
Use the **ResNet-50** as the backbone instead.

	50-layer
	7×7, 64, stride
	3×3 max pool, stri
block1	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
block2	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$
block3	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$
block4	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$

unit



Each block contains several units.



Convert the TF graph to the TRT graph directly

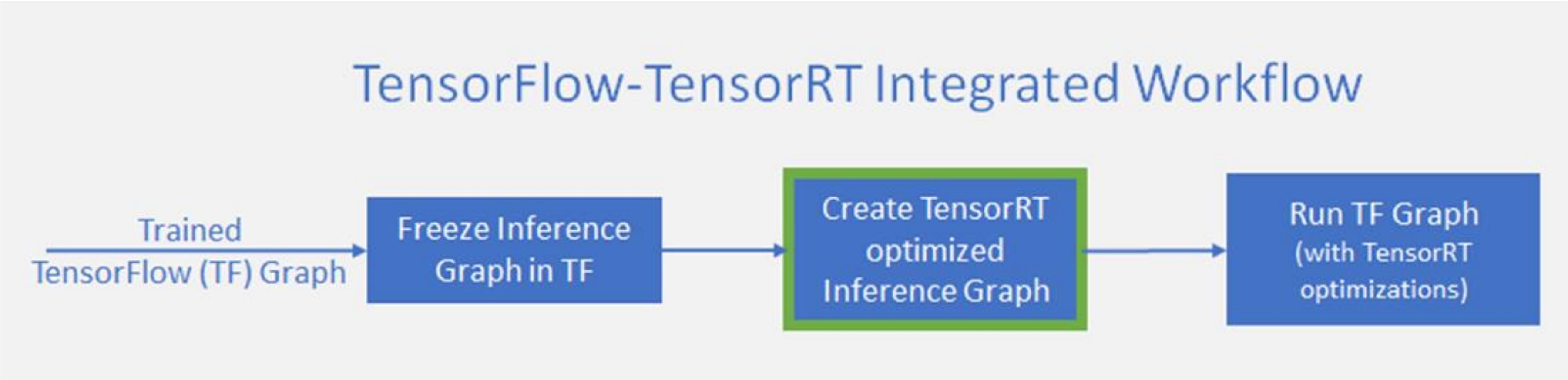
Create the network from scratch

Parse the network from the TF model

TFTRT

TFTRT (TensorFlow integration with TensorRT) parses the frozen TF graph and **converts each supported subgraph to a TRT optimized node** (TRTEngineOp), allowing TF to execute the remaining graph.

Create a frozen graph from a trained TF model, and give it to the Python API of TF-TRT.



https://docs.nvidia.com/deeplearning/dgx/tf-trt-user-guide/index.html

SETUP

Install:

TFTRT is part of the TensorFlow binary, which means when you install tensorflow-gpu, you will be able to use TF-TRT too. (**pip install tensorflow-gpu**)

prerequisite:

import modules

```
import tensorflow as tf
import tensorflow.contrib.tensorrt as trt
```

the names of input and output nodes

```
inputs = "input_images"
outputs = ["feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"]
```

the TF model trained in FP32 (checkpoint or pb files)

```
model_infer.ckpt-49491.data-00000-of-00001
model_infer.ckpt-49491.index
model_infer.ckpt-49491.meta
```

Step 1 Obtain the TF frozen graph

- **With Ckpt**

```
with tf.Session( ) as sess:
    # Import the “MetaGraphDef” protocol buffer, and restore the variables
    saver = tf.train.import_meta_graph("model.ckpt.meta")
    saver.restore(sess, "model.ckpt")
    # freeze the graph (convert all Variable ops to Const ops holding the same values)
    outputs = ["feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"] #node names
    frozen_graph = tf.graph_util.convert_variables_to_constants(sess, sess.graph_def,
                  output_node_names=outputs)
```

- **With Pb**

```
with tf.Session( ) as sess:
    # deserialize the frozen graph
    with tf.gfile.Gfile("./model.pb", "rb") as f:
        frozen_graph = tf.GraphDef()
        frozen_graph.ParseFromString(f.read())
```

https://docs.nvidia.com/deeplearning/dgx/tf-trt-user-guide/index.html

Step 2 Create the TRT graph from the TF frozen graph

```
trt_graph = trt.create_inference_graph (
        input_graph_def = frozen_graph, outputs = output_node_name,
        max_batch_size = 1, max_workspace_size_bytes = 1<<30,
        precision_mode = ="FP32",
        minimum_segment_size = 5, … )
```

input_graph_def: the frozen TF GraphDef object

outputs: the names list of output nodes

max_batch_size: maximum batch size

max_workspace_size_bytes: maximum GPU memory size available for TRT layers

precision_mode: FP32 / FP16 / INT8

minimum_segment_size: determine the minimum number of nodes in a TF sub-graph for the TRT engine to be created

https://docs.nvidia.com/deeplearning/dgx/tf-trt-user-guide/index.html

Step 3 Import the TRT graph and run

```
# import the TRT graph into the current default compute graph
g = tf.get_default_graph()
inputs= g.get_tensor_by_name("input_images:0")
outputs = [n+':0' for n in outputs] # tensor names
f_score, f_geo = tf.import_graph_def(trt_graph, input_map={"input_images": inputs},
                 return_elements=outputs, name="")

# run the optimized graph in session
img = cv2.imread("xxx.jpg")
score, geometry = sess.run([f_score, f_geo], feed_dict={inputs: [img]})
```

https://docs.nvidia.com/deeplearning/dgx/tf-trt-user-guide/index.html

TFTRT FP32

```
with tf.Session( ) as sess:
    # create a `Saver` object, import the “MetaGraphDef” protocol buffer, and restore the variables
    saver = tf.train.import_meta_graph("model.ckpt.meta")
    saver.restore(sess, "model.ckpt")
    # freeze the graph (convert all Variable ops to Const ops holding the same values)
    outputs = ["feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"] #node names
    frozen_graph = tf.graph_util.convert_variables_to_constants(sess, sess.graph_def,
                   output_node_names=outputs)

    # create a TRT inference graph from the TF frozen graph
    trt_graph = trt.create_inference_graph(input_graph_def=frozen_graph, outputs=outputs,
                max_batch_size=1, max_workspace_size_bytes=1<<30,
                precision_mode="FP32",
                minimum_segment_size=5)
    # import the TRT graph into the current default graph
    g = tf.get_default_graph()
    input_images = g.get_tensor_by_name("input_images:0")
    outputs = [n+':0' for n in outputs] # tensor names
    f_score, f_geometry = tf.import_graph_def(trt_graph, input_map={"input_images":input_images},
                return_elements=outputs, name="")

    # run the optimized graph in session
    img = cv2.imread("./img.jpg")
    score, geometry = sess.run([f_score, f_geometry], feed_dict={input_images: [img]})
```

https://docs.nvidia.com/deeplearning/dgx/tf-trt-user-guide/index.html

TFTRT FP16

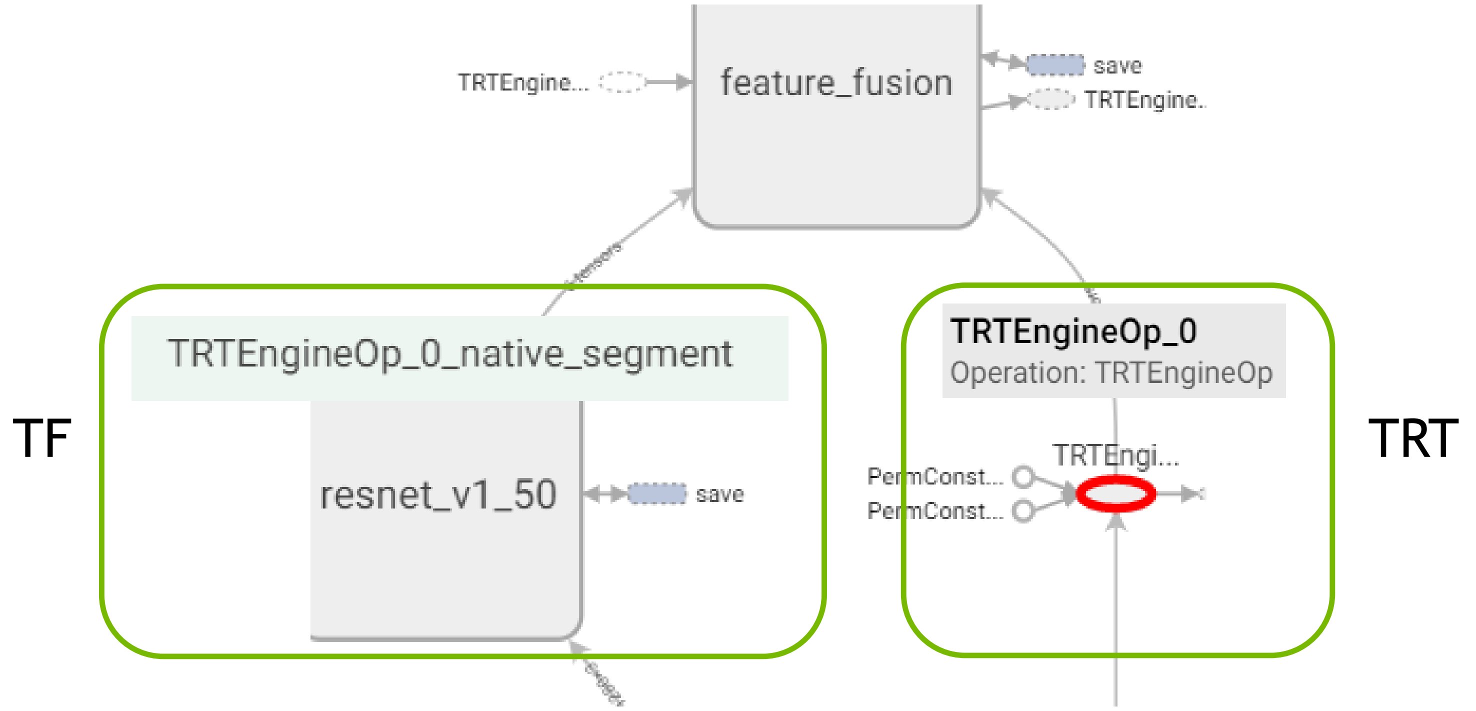
```
with tf.Session( ) as sess:
    # create a `Saver` object, import the “MetaGraphDef” protocol buffer, and restore the variables
    saver = tf.train.import_meta_graph("model.ckpt.meta")
    saver.restore(sess, "model.ckpt")
    # freeze the graph (convert all Variable ops to Const ops holding the same values)
    outputs = ["feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"] #node names
    frozen_graph = tf.graph_util.convert_variables_to_constants(sess, sess.graph_def,
                        output_node_names=outputs)

    # create a TRT inference graph from the TF frozen graph
    trt_graph = trt.create_inference_graph(input_graph_def=frozen_graph, outputs=outputs,
                    max_batch_size=1, max_workspace_size_bytes=1<<30,
                    precision_mode="FP16",
                    minimum_segment_size=5)
    # import the TRT graph into the current default graph
    g = tf.get_default_graph()
    input_images = g.get_tensor_by_name("input_images:0")
    outputs = [n+':0' for n in outputs] # tensor names
    f_score, f_geometry = tf.import_graph_def(trt_graph, input_map={"input_images":input_images},
                    return_elements=outputs, name="")

    # run the optimized graph in session
    img = cv2.imread("./img.jpg")
    score, geometry = sess.run([f_score, f_geometry], feed_dict={input_images: [img]})
```

https://docs.nvidia.com/deeplearning/dgx/tf-trt-user-guide/index.html

Visualize the Optimized Graph in TensorBoard

```
Engine resnet_v1_50/my_trt_op_0 creation for segment 0, composed of 446 nodes succeeded
```

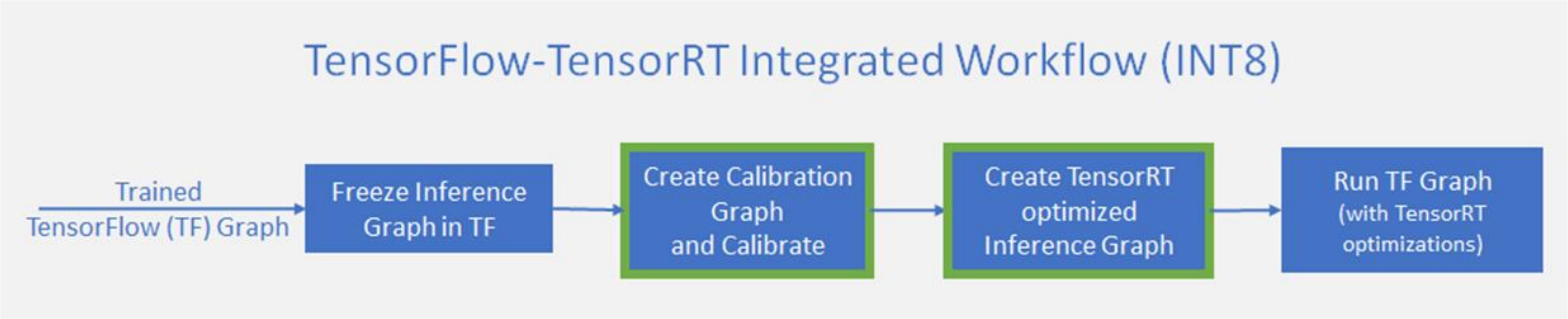
```
Total node count before and after TF-TRT conversion: 900 -> 165
```



TFTRT converts the native TF subgraph (TRTEngineOp_0_native_segment) to a single TRT node (TRTEngineOp_0).

TFTRT INT8

The INT8 precision mode requires an additional **calibration step** before quantization.



INT8_value = FP32_value * scale

Calibration: run inference in FP32 precision on a calibration dataset, which collects required statistics and runs the calibration algorithm, to generate INT8 quantization (scaling factors) of the weights and activations in the trained TF graph.

http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf

TFTRT INT8

Step 1 Obtain the TF frozen graph (trained in FP32)

…

Step 2 Create the calibration graph -> Execute it with calibration data -> Convert it to the INT8 optimized graph

```
# create a TRT inference graph, the output is a frozen graph ready for calibration
calib_graph = trt.create_inference_graph(input_graph_def=frozen_graph, outputs=outputs,
              max_batch_size=1, max_workspace_size_bytes=1<<30,
              precision_mode="INT8", minimum_segment_size=5)

# Run calibration (inference) in FP32 on calibration data (no conversion)
f_score, f_geo = tf.import_graph_def(calib_graph, input_map={"input_images":inputs},
              return_elements=outputs, name="")
Loop img: score, geometry = sess.run([f_score, f_geo], feed_dict={inputs: [img]})

# apply TRT optimizations to the calibration graph, replace each TF subgraph with a TRT node
optimized for INT8
trt_graph = trt.calib_graph_to_infer_graph(calib_graph)
```

Step 3 Import the TRT graph and run

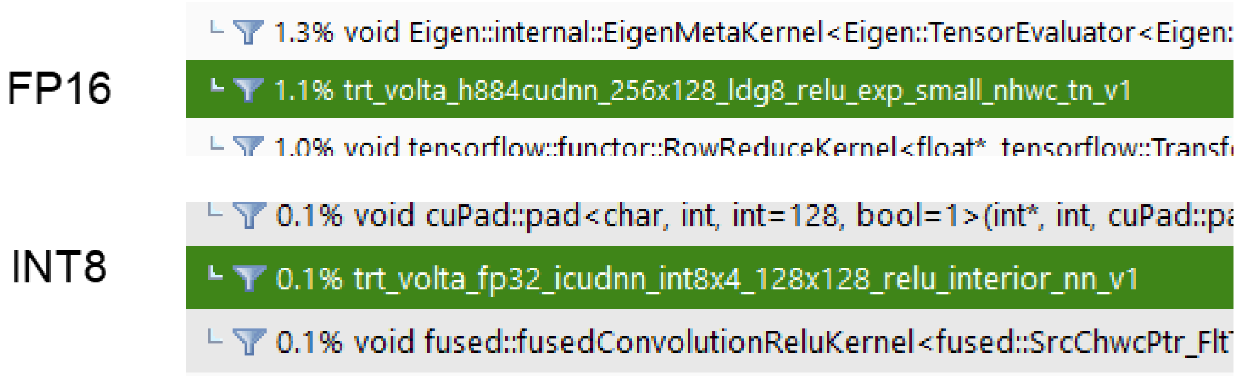
…

https://docs.nvidia.com/deeplearning/dgx/tf-trt-user-guide/index.html

TFTRT FP32/FP16/INT8 Performance (V100, batch size = 1)

ICDAR2015 TestSet (672x1280)	FPS	recall	precision	F1score
TF Slim	42	0.7732	0.8466	0.8083
TFTRT FP32	63	0.7732	0.8466	0.8083
TFTRT FP16	98	0.7723	0.8442	0.8066
TFTRT INT8	83	0.7602	0.8572	0.8058

INT8 with IDP.4A instruction is slower than FP16 with Tensor Core on V100.



h884cudnn: HMMA for Volta, fp16 input, output, and accumulator.

fp32_icudnn_int8x4: Int8 kernels using the IDP.4A instruction. Inputs are aligned to fetch 4x int8 in one instruction.

https://docs.google.com/spreadsheets/d/1xAo6TcSgHdd25EdQ-6GqM0VKbTYu8cWyycgJhHRVIgY/edit#gid=1454841244

TAKEAWAYS

- The names of input and output nodes

```
inputs = "input_images"
outputs = ["feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"]
```

- The TF model trained in FP32 (checkpoint or pb files)

```
model_infer.ckpt-49491.data-00000-of-00001
model_infer.ckpt-49491.index
model_infer.ckpt-49491.meta
```

- Calibration dataset for INT8 quantization

```
img_113.jpg  img_159.jpg  img_203.jpg  img_249.jpg
img_114.jpg  img_15.jpg   img_204.jpg  img_24.jpg
img_115.jpg  img_160.jpg  img_205.jpg  img_250.jpg
```

Tips 1: GPU memory allocation

Specify the fraction of GPU memory allowed for TF, making the remaining available for TRT engines.

Use the **per_process_gpu_memory_fraction** and **max_workspace_size_bytes** parameters together for best overall application performance.

```
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.6)
tf_config = tf.ConfigProto(gpu_options=gpu_options, allow_soft_placement=True)
with tf.Session(config=tf_config) as sess:
```

```
trt_graph = trt.create_inference_graph(input_graph_def=frozen_graph, outputs=outputs,
        max_batch_size=1,
        max_workspace_size_bytes=1<<30,
        precision_mode="FP32",
        minimum_segment_size=5)
```

Certain algorithms in TRT need a larger workspace, therefore, decreasing the TF-TRT workspace size might result in not running the fastest TRT algorithms possible.

Tips 2: Minimum segment size

To achieve the best performance, different possible values of minimum_segment_size can be tested.

We can start by setting it to a large number and decrease this number until the converter crashes.

min_seg_size	TRT nodes
50	1 (446 tf nodes)
30	2 (44 / 446 tf nodes)
5	4 (24 / 24 / 44 / 446 tf nodes)
3	5 (4 / 24 / 24 / 44 / 446 tf nodes)

```
trt.create_inference_graph (…, minimum_segment_size = 5, … )
```

determine the minimum number of nodes in a TF sub-graph for the TRT engine to be created

Tips 3: Batch normalization

The FusedBatchNorm operator is converted to TRT only if is_training=False, which indicates whether the operation is for training or inference.

```
(Pdb) frozen_graph.node[176]
name: "model_0/resnet_v1_50/block1/unit_2/bottleneck_v1/conv1/BatchNorm/cond/FusedBatchNorm"
op: "FusedBatchNorm"
attr {
  key: "is_training"
  value {
    b: true
  }
```

tf.train.import_meta_graph("xxx.ckpt") just imports the saved graph, usually training graph.

Need to change the **is_training=False** in the graph.

https://nvbugswb.nvidia.com/NvBugs5/SWBug.aspx?bugid=200503725&cmtNo=

Tips 3: Batch normalization

Workarounds:

1. **With the codes** building the network:

Build the TF inference graph by setting **is_training=false** for all fusedBatchNorm layers, and then restore the weights from the training graph without using tf.train.import_meta_graph.

```
input_images = tf.placeholder(tf.float32, shape=[None, 672, 1280, 3],
                              name='input_images')
f_score, f_geometry = model.model(input_images, is_training=False)
saver = tf.train.Saver()
with tf.Session(config=tf_config) as sess:
    saver.restore(sess, "./model.ckpt")
    tf.graph_util.convert_variables_to_constants...
    trt.create_inference_graph...
```

```
with slim.arg_scope([slim.batch_norm], is_training=is_training):
```

https://nvbugswb.nvidia.com/NvBugs5/SWBug.aspx?bugid=200503725&cmtNo=

Tips 3: Batch normalization

Workarounds:

2. **Without the codes** building the network:

   Resave an inference graph as the ckpt files and then use the **tf.train.import_meta_graph** API directly.

Customer provided:

```
input_images = tf.placeholder(tf.float32, shape=[None, 672, 1280, 3],
                              name='input_images')
f_score, f_geometry = model.model(input_images, is_training=False)
saver = tf.train.Saver()
with tf.Session(config=tf_config) as sess:
    saver.restore(sess, "./model.ckpt")
    saver.save(sess, './model_infer.ckpt')
```

Then

```
saver = tf.train.import_meta_graph("model_infer.ckpt.meta")
saver.restore(sess, "model_infer.ckpt")
```

https://nvbugswb.nvidia.com/NvBugs5/SWBug.aspx?bugid=200503725&cmtNo=

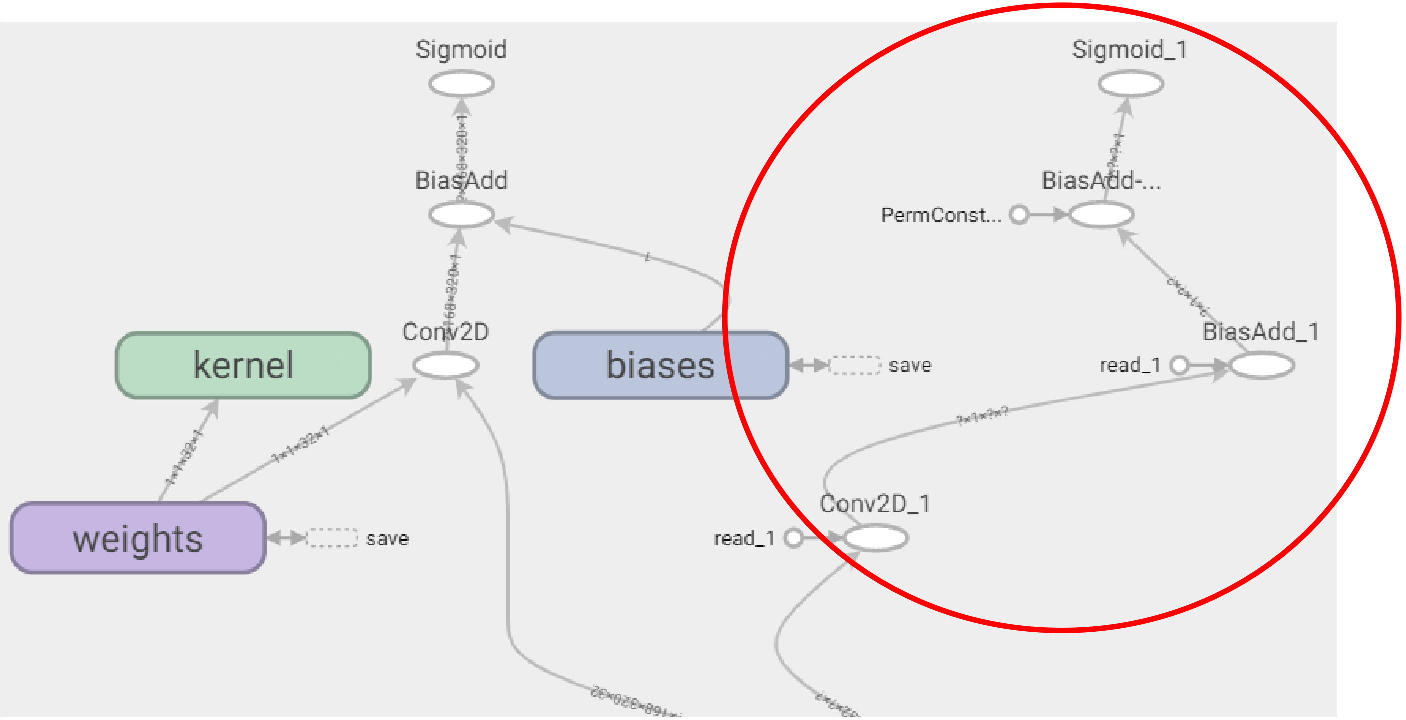
Tips 4: TRT node name

```
…
tf.import_graph_def(trt_graph, …)
g = tf.get_default_graph()
f_score = g.get_tensor_by_name("Conv_7/Sigmoid_1:0")
…
```

⟺

```
f_score = tf.import_graph_def(…)
```

```
score = sess.run([f_score], feed_dict={inputs: [img]})
```



if the same name has been previously used in the same scope, it will be **made unique by appending _N** to it.

https://www.tensorflow.org/api_docs/python/tf/variable_scope

TRT API

Install the TensorRT SDK

import tensorrt as trt

Load all weights from the saved model (the TF model trained in FP32)

↓

Create the network from scratch with TRT layer APIs (the network structure)

↓

Build the TRT engine

↓

Create the context and execute inference

https://docs.nvidia.com/deeplearning/sdk/tensorrt-api/python_api/index.html

TAKEAWAYS

- The TF model trained in FP32 (checkpoint or pb files)

```
model_infer.ckpt-49491.data-00000-of-00001
model_infer.ckpt-49491.index
model_infer.ckpt-49491.meta
```

- The details of network (names and shapes of all weights, network structure, etc.)

Codes building the network if possible

or

Visualize the network in TensorBoard

```
with tf.Session(config=tf.ConfigProto(allow_soft_placement=True)) as sess:
        saver = tf.train.import_meta_graph("model_infer.ckpt-98981.meta")
        saver.restore(sess, "model_infer.ckpt-98981")
        summary_writer = tf.summary.FileWriter('./log/', sess.graph)
```

events.out.tfevents.1559529132.dgxstation49.nvidia.com

TRT API

Step 1 Load all learned weights from the saved model

```
reader = tf.train.NewCheckpointReader("./model.ckpt-98981")
```

Step 2 Create the network from scratch with TRT layer APIs, and build the engine

```
with trt.Builder(G_LOGGER) as builder, builder.create_network() as network:
    data = network.add_input("data", trt.float32, (3, input_h, input_w)) # add the input layer
    # add the convolution layer
    w = reader.get_tensor("resnet_v1_50/conv1/weights")
    conv = network.add_convolution(data, out_channel, (kernel_h,kernel_w), trt.Weights(w), trt.Weights(b))
    conv.stride = (stride, stride); conv.padding = (padding, padding)
    …
    network.mark_output(outputs.get_output(0)) # mark outputs
    engine = builder.build_cuda_engine(network) # build the engine
```

Step 3 Create the context and execute inference

```
# The TF’s input [NHWC] should be transposed to TRT format [NCHW]
with engine.create_execution_context() as context:
    [cuda.memcpy_htod(inp.device, inp.host) for inp in inputs]
    context.execute(batchsize, bindings)
    [cuda.memcpy_dtoh(out.host, out.device) for out in outputs]
```

https://docs.nvidia.com/deeplearning/sdk/tensorrt-api/python_api/index.html

Tips 1: Tensor format

TensorFlow [**NHWC**] → TensorRT [**NCHW**]

The TF’s input should be transposed to TRT’s explicitly, so is the output.

```
im = cv2.imread("test.jpg")[:, :, ::-1]
img, (ratio_h, ratio_w) = resize_image(im)
img = img.astype(np.float32)
img = mean_image_subtraction(img)
img = np.transpose(img, (2, 0, 1))
img = np.array([img])

with engine.create_execution_context() as context:
    # execute inference
    img = img.ravel()
    np.copyto(inputs[0].host, img)
    [cuda.memcpy_htod(inp.device, inp.host) for inp in inputs]
    context.execute(batchsize, bindings)
    [cuda.memcpy_dtoh(out.host, out.device) for out in outputs]
```

Tips 2: Weight format

CONV: TensorFlow [**RSCK**] → TensorRT [**KCRS**]

RSCK: [filter_height, filter_width, in_channel, out_channel]

KCRS: [out_channel, in_channel, filter_height, filter_width]

```
w = reader.get_tensor("resnet_v1_50/conv1/weights")
w = w.transpose(3,2,0,1).reshape(-1) #RSCK->KCRS
b = np.zeros(out_channel, dtype=np.float32)
conv = network.add_convolution(inputs, out_channel, (kernel_size,kernel_size),
                               trt.Weights(w), trt.Weights(b))
conv.stride = (stride, stride)
conv.padding = (padding, padding)
```

FC: TensorFlow [**CK**] → TensorRT [**KC**]

Tips 3: SAME padding

SAME padding in TF may lead to asymmetric padding.

```
net = slim.max_pool2d(net, [3, 3], stride=2, padding='SAME', scope='pool1')
```

Input map (one channel) : 336x640 → Output map (one channel) : 168x320

$$h\_output = (h\_input - h\_kernel + h\_pad) // h\_stride + 1 \rightarrow 168 = (336 - 3 + 1) // 2 + 1$$

```
               pad|                                           |pad
inputs:         0 |1  2  3  4  5  6  7  8  9  10 11 12 13|0  0
               |________________|
                              |_________________|
                                             |________________|
```

```
top = network.add_pooling(top.get_output(0), trt.PoolingType.MAX, (3,3))
top.stride = (2,2)
top.pre_padding = (0,0)
top.post_padding = (1,1)
```

Tips 4: Batch normalization

$$bn(x) = \frac{x - mean}{\sqrt{var + \varepsilon}} * gamma + beta \longrightarrow output = (input * scale + shift)^{power}$$

$$scale = \frac{gamma}{\sqrt{var + \epsilon}} \qquad shift = -\frac{mean}{\sqrt{var + \epsilon}} * gamma + beta \qquad power = 1$$

```python
# load gamma, beta, moving_mean and moving variance with CKPT reader
gamma = reader.get_tensor("resnet_v1_50/conv1/BatchNorm/gamma")
beta = reader.get_tensor("resnet_v1_50/conv1/BatchNorm/beta")
mean = reader.get_tensor("resnet_v1_50/conv1/BatchNorm/moving_mean")
var = reader.get_tensor("resnet_v1_50/conv1/BatchNorm/moving_variance")
# calculate the parameters and apply the scale layer
scale = gamma / np.sqrt(var + 1e-5)
shift = -mean / np.sqrt(var + 1e-5) * gamma + beta
power = np.ones(out_channel, dtype=np.float32)
bn = network.add_scale(conv.get_output(0), trt.ScaleMode.CHANNEL, trt.Weights(shift),
                      trt.Weights(scale), trt.Weights(power))
```

TRT UFF PARSER

Step 1 Convert the pb model to the uff model

```
convert-to-uff model.pb
```

Step 2 Parse the uff model and create the engine

```
with trt.Builder(G_LOGGER) as builder, builder.create_network() as network, trt.UffParser() as parser:
    builder.max_batch_size = 1
    builder.max_workspace_size = 1<<30
    parser.register_input("input_images", (3, 672, 1280))
    parser.register_output("feature_fusion/Conv_7/Sigmoid")
    parser.register_output("feature_fusion/concat_3")
    parser.parse("./model.uff", network)
    engine = builder.build_cuda_engine(network)
```

Step 3 Create context and execute inference

```
# The TF's input [NHWC] should be transposed to TRT format [NCHW], no need for output
with engine.create_execution_context() as context:
    [cuda.memcpy_htod(inp.device, inp.host) for inp in inputs]
    context.execute(batchsize, bindings)
    [cuda.memcpy_dtoh(out.host, out.device) for out in outputs]
```

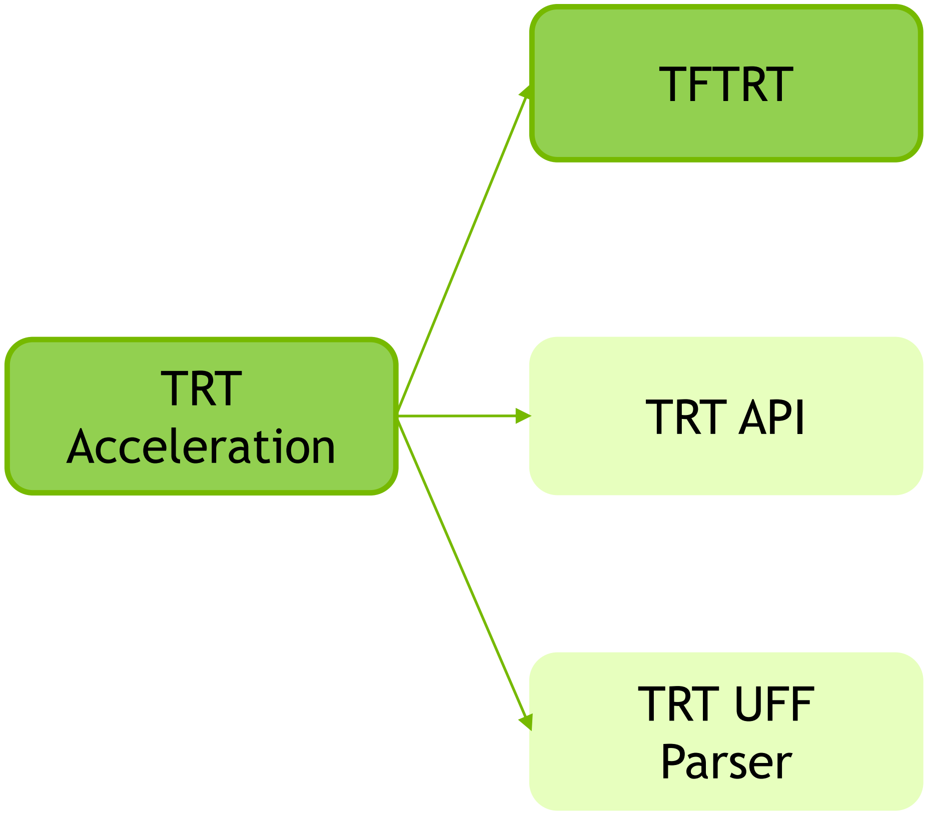
PERFORMANCE

V100, FP32, ICDAR2015 TestSet 672x1280

Batchsize	TF Slim	TFTRT	TRT API	TRT Parser
1	48.4	62.15	75.02	75.23
4	57.78	73.88	85.13	85.45
16	63.57	77.18	88.47	88.18

- Increasing batchsize (up to 16) improves FPS on single V100.
- TRT API and TRT Parser are more efficient in FPS than TFTRT here.
- The performances of TRT API and TRT Parser are almost the same.

CONCLUSION



- TFTRT is easy and convenient to use for TF model, but with limited acceleration now.
- TRT API and TRT UFF Parser are able to achieve better performance than TFTRT.
- TRT UFF Parser is constrained by supported ops in TRT unless adding plugins.
- TRT API is more flexible to create the network, but may lead to more work.

RESOURCES

- EAST: An Efficient and Accurate Scene Text Detector
  https://arxiv.org/abs/1704.03155
- EAST implement in TF
  https://github.com/argman/EAST
- TFTRT user guide
  https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html
- TRT developer guide
  https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html
- TRT API guide
  https://docs.nvidia.com/deeplearning/sdk/tensorrt-api/index.html

Acknowledgement:
Chandler Zhou, Gary Ji, Xipeng Li @ Devtech,
and Rita Zhang.